Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–III EXAMINATION – SUMMER 2025

Subject Code:3130702 Date:29-05-2025

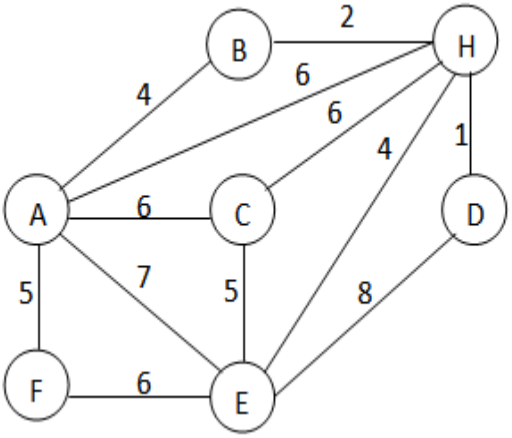
Subject Name:Data Structures

Time:02:30 PM TO 05:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

Q.1	**(a)**	What is the best-case, average-case, and worst-case time complexity analysis?	**03**
	(b)	Explain row-major order and column-major order representation of 2-D array.ss	**04**
	(c)	Construct a Binary Search Tree for the following data. 21, 51, 12, 45, 17, 71, 19, 47, 78. Write Pre-order, In-order, and Post-order traversal of constructed BST.	**07**
Q.2	**(a)**	Define following terms: 1. Full Binary Tree 2. Complete Binary Tree 3. Skewed Binary Tree	**03**
	(b)	Write the importance of asymptotic analysis. Is $O(n \log_2 n^2)$ faster than $O(n^2)$? Justify your answer with an example.	**04**
	(c)	Convert the following infix expression into a postfix expression using stack. (a+b)^((c*d)/(e-f))	**07**
		OR	
	(c)	Write an algorithm to convert infix expression into postfix expression.	**07**
Q.3	**(a)**	Illustrate how stack is used in the recursion.	**03**
	(b)	Describe Threaded Binary Tree with example.	**04**
	(c)	Write a C program for the following operations on a circular queue. 1. Insert 2. Delete 3. Display	**07**
		OR	
Q.3	**(a)**	Write a recursive solution for Tower of Hanoi problem.	**03**
	(b)	Explain the DFS traversal of the graph with an example.	**04**
	(c)	Write an algorithm to sort existing singly linked list in acceding order according to the information field.	**07**
Q.4	**(a)**	Define the following terms: 1. Field 2. Record 3. File	**03**
	(b)	Sort the following data using merge sort. 50, 20, 70, 05, 30, 80, 55, 25	**04**
	(c)	Find the minimum spanning tree of the following graph using Kruskal's algorithm.	**07**

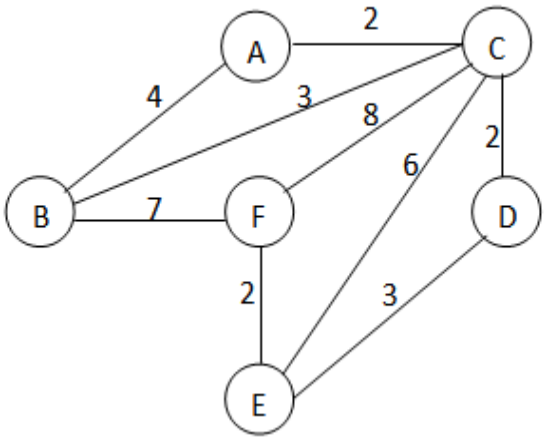


OR

Q.4 **(a)** What is hashing? Write the properties of a good hash function. **03**

(b) Sort the following data using quick sort. **04**
50, 30,80,40, 35,70, 60, 20,75

(c) Find the shortest path from A to F using Dijkstra's Algorithm. **07**



Q.5 **(a)** Compare linear search and binary search in terms of their time complexity. **03**

(b) Write a C program for a bubble sort. **04**

(c) What is hash collision? Explain collision resolution techniques. **07**

OR

Q.5 **(a)** Does a pivot selection method affect the time complexity of quick sort? Justify your answer. **03**

(b) Write a C program for a selection sort. **04**

(c) List various file organizations and explain one in detail. **07**

****

Q.1 (a) What is the best-case, average-case, and worst-case time complexity analysis? (3 mark)

Introduction

Time complexity analysis helps us **measure the efficiency of an algorithm** in terms of the number of steps or operations it takes to complete. It is usually expressed in **Big O notation** and is analyzed under three scenarios: **best-case, average-case, and worst-case**. This analysis is essential for understanding how an algorithm performs under different conditions and helps in choosing the most suitable algorithm for a problem.

---

1. Best-Case Time Complexity

- The **best-case** represents the scenario where the algorithm performs the **minimum number of steps**.
- It occurs when the input is **most favorable**.
- Example: For **linear search**, if the target element is the first element in the list, the algorithm stops immediately.
- Purpose: Helps to know the **minimum time required** by the algorithm.

---

2. Average-Case Time Complexity

- The **average-case** represents the **expected time** an algorithm takes for a typical input.
- It considers **all possible inputs** and calculates the average number of steps.
- Example: In **linear search**, if the target element is equally likely to be at any position, on average the algorithm searches **half of the list**.

- Purpose: Provides a **realistic estimate** of performance in practical scenarios.

---

3. Worst-Case Time Complexity

- The **worst-case** represents the scenario where the algorithm takes the **maximum number of steps**.
- It occurs when the input is **least favorable**.
- Example: In **linear search**, if the target element is not present or is the last element, all elements are checked.
- Purpose: Helps to ensure that the algorithm **will not exceed a certain time limit** under any circumstance.

(b) Explain row-major order and column-major order representation of 2-D array. (4 mark)

Introduction

A **2-D array** is stored in **linear memory** (1-D memory) because computer memory is essentially sequential. To map a 2-D array into 1-D memory, we use **row-major order** or **column-major order**. These methods determine how elements of a 2-D array are stored and accessed in memory.

---

1. Row-Major Order

- In **row-major order**, the elements of each **row** are stored **contiguously** in memory.
- The next row follows immediately after the previous row.
- Formula for the address of an element **A[i][j]** (assuming base address = BA, number of columns = n):

$$\text{Address}(A[i][j]) = BA + [(i \times n) + j] \times \text{size of element}$$

- **Example:**
  Consider a 2×3 array:

  1 2 3

  4 5 6
- **Row-major storage in memory:** 1, 2, 3, 4, 5, 6
- **Use:** Common in languages like **C**.

---

2. Column-Major Order

- In **column-major order**, the elements of each **column** are stored **contiguously** in memory.
- The next column follows immediately after the previous column.
- Formula for the address of an element **A[i][j]** (assuming base address = BA, number of rows = m):
- Address(A[i][j])=BA+[(j×m)+i]×size of element
- **Example:**
  Consider the same 2×3 array:

  Column-major storage in memory: 1, 4, 2, 5, 3, 6
- **Use:** Common in languages like **Fortran** and MATLAB.

---

3. Key Differences

Feature	Row-Major Order	Column-Major Order
Storage	Row-wise	Column-wise
Memory access	Contiguous elements in a row	Contiguous elements in a column
Formula	`Address = BA + [(i*n)+j]*size`	`Address = BA + [(j*m)+i]*size`
Example languages	C, C++	Fortran, MATLAB

(c) Construct a Binary Search Tree for the following data. 21, 51, 12, 45, 17, 71, 19, 47, 78. Write Pre-order, In-order, and Post-order traversal of constructed BST.(7 mark)

Q.1 (c) Construct a Binary Search Tree (BST) and Write Its Traversals

Introduction

A **Binary Search Tree (BST)** is a binary tree where:

- The **left subtree** of a node contains elements **less than the node**.
- The **right subtree** contains elements **greater than the node**.
- Both left and right subtrees are themselves BSTs.

BSTs allow **efficient searching, insertion, and deletion** operations with an average time complexity of O(log n). We are given the data: **21, 51, 12, 45, 17, 71, 19, 47, 78**

We will construct the BST step by step.

Step 1: Constructing the BST

1. **Insert 21:**
    - First element becomes the root.

2. 21

3. **Insert 51:**
    - 51 > 21 → goes to the **right**.

4. 21

5. \

6. 51

7. **Insert 12:**

- 12 < 21 → goes to the **left**.

8. 21
9. / \
10. 12 51
11. **Insert 45:**
    - 45 < 51 → left of 51
    - 45 > 21 → right of 21
12. 21
13. / \
14. 12 51
15. /
16. 45
17. **Insert 17:**
    - 17 > 12 → right of 12
    - 17 < 21 → left of 21
18. 21
19. / \
20. 12 51
21. \
22. 17
23. \
24. **Insert 71:**
    - 71 > 51 → right of 51
25. 21

26. / \

27. 12 51

28. \ / \

29. 17 45 71

30. **Insert 19:**

    - 19 > 17 → right of 17

31. 21

32. / \

33. 12 51

34. \

35. 17

36. \

37. 19

38. / \

39. 45 71

40. **Insert 47:**

    - 47 > 45 → right of 45
    - 47 < 51 → left of 51

41. **Insert 78:**

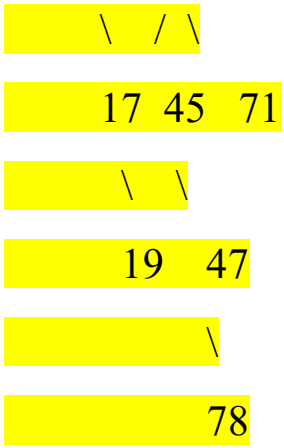
    - 78 > 71 → right of 71

Final BST Structure

21

/ \

12 51



Step 2: Traversals of BST

1. Pre-order Traversal (Root → Left → Right)

- Visit Root → Traverse Left Subtree → Traverse Right Subtree
- **Result:** 21, 12, 17, 19, 51, 45, 47, 71, 78

---

2. In-order Traversal (Left → Root → Right)

- Visit Left Subtree → Root → Right Subtree
- **Result:** 12, 17, 19, 21, 45, 47, 51, 71, 78
- **Observation:** In-order traversal of a BST gives **sorted order**.

---

3. Post-order Traversal (Left → Right → Root)

- Visit Left Subtree → Right Subtree → Root
- **Result:** 19, 17, 12, 47, 45, 78, 71, 51, 21

Q.2 (a) Define following terms: 1. Full Binary Tree 2. Complete Binary Tree 3. Skewed Binary Tree

A **Binary Tree** is a hierarchical data structure in which each node has at most **two children**: left and right. Based on the arrangement of nodes, binary trees can have various forms. Three important types are:

---

1. Full Binary Tree

- A **Full Binary Tree** is a binary tree in which **every node has either 0 or 2 children**.
- No node has only one child.
- **Characteristics:**
  - Each level, except possibly the last, is completely filled.
  - Also called a **proper binary tree**.
- **Example:**

```
    10
   /  \
  20   30
```

- Here, each node has 0 or 2 children.

---

2. Complete Binary Tree

- A **Complete Binary Tree** is a binary tree in which **all levels are completely filled except possibly the last level**, and the last level has all nodes **left-aligned**.
- **Characteristics:**
  - Efficiently stored in arrays.
  - Used in **heap data structures**.
- **Example:**

```
      10
     /  \
    20   30
   /  \
  40  50
```

- Last level nodes (40, 50) are filled from the left.

---

3. Skewed Binary Tree

- A **Skewed Binary Tree** is a tree in which **all nodes have only one child**.
- It can be of two types:
  - **Left-skewed:** every node has only left child
  - **Right-skewed:** every node has only right child
- **Characteristics:**
  - Essentially behaves like a **linked list**.
  - Used in cases where **linear data storage** is required.
- **Example (Right-skewed):**

10 → 20 → 30 → 40

(b) Write the importance of asymptotic analysis. Is O(n log2 n2) faster than O(n2)? Justify your answer with an example. (4 mark)

Definition / Introduction

Asymptotic analysis is a mathematical method used to evaluate the **efficiency of an algorithm** by studying how its running time or space requirement grows with the **input size (n)**. Instead of measuring actual time, asymptotic analysis focuses on the **growth rate** of an algorithm

for large inputs, which helps compare algorithms independently of hardware or programming language.

Importance of Asymptotic Analysis

The importance of asymptotic analysis can be explained as follows:

1. Machine-Independent Performance Measurement

- It evaluates the speed of an algorithm **without depending on CPU, compiler, or machine architecture**.
- It focuses only on how the algorithm behaves mathematically.

2. Helps Compare Multiple Algorithms

- For the same problem, asymptotic notation helps identify which algorithm performs better as input size increases.
- Example: Comparing **O(log n)**, **O(n)**, and **O(n²)**.

3. Predicts Performance for Large Inputs

- For real-world applications (sorting, searching, graphs), input size can be very large.
- Asymptotic analysis helps understand the algorithm's feasibility.

4. Simplifies Running Time Expression

- Removes constant factors and lower-order terms.
- Provides a cleaner form like **O(n log n)** instead of complex stepwise counts.

5. Helps in Algorithm Optimization

- Identifies bottlenecks.
- Allows designers to improve logic and data structures for efficiency.

Is O(n $\log_2 n^2$) Faster than O(n^2)?

First simplify:

Step 1: Simplify the expression

We know:

$$\log_2(n^2) = 2\log_2 n$$

So,

$$O(n\log_2 n^2) = O(n \cdot 2\log_2 n$$

And we ignore constant 2:

$$O(n\log\ n)$$

Step 2: Compare O(n log n) and O(n^2)

- **O(n log n)** grows **much slower** than **O(n^2)** as n increases.
- Therefore,

✅ **O(n log n) is faster than O(n^2).**

Example to Justify

Let's compare both for some sample values:

n	**n $\log_2$ n**	**n^2**
10	10 × 3.3 = 33	100
100	100 × 6.6 = 660	10,000

n	$n \log_2 n$	n^2
1000	$1000 \times 10 = 10{,}000$	1,000,000

- For **n = 1000**,
  - **O(n log n)** ≈ 10,000
  - **O(n²)** = 1,000,000
- Clearly, **O(n log n)** takes far fewer operations.

Thus, **O(n $\log_2$ n²)** (which simplifies to **O(n log n)**) is **faster** than **O(n²)** for large inputs.

(c) Convert the following infix expression into a postfix expression using stack. (a+b)^((c*d)/(e-f)) (7 mark)

Introduction

Infix notation is the common mathematical notation where operators are written **between operands**, such as A + B. However, computers find it easier to evaluate expressions written in **postfix (Reverse Polish Notation)** because it does not require parentheses and follows a clear operator precedence using stacks.

To convert an infix expression to postfix, we use a **stack-based algorithm** that processes operands, operators, and parentheses in a systematic way. In this answer, we will convert the given expression using step-by-step stack operations, operator precedence, and associativity rules.

Rules Used for Conversion

1. If the symbol is an **operand**, add it directly to postfix.
2. If the symbol is an **operator**,

- Pop from the stack until top has an operator of **lower precedence**,
- Then push the current operator.

3. If the symbol is **(**, push it.
4. If the symbol is **)**, pop all operators until **(** is found.
5. At the end, pop all operators from the stack.

Operator precedence (high → low):

1. ^ (right associative)
2. *, /
3. +, –

Step-by-Step Conversion Using Stack

Expression:

(a + b) ^ ((c * d) / (e – f))

We process it left to right:

Step 1: (

Stack: (
Postfix: —

Step 2: a

Stack: (
Postfix: a

Step 3: +

Stack: (+
Postfix: a

Step 4: b

Stack: (+
Postfix: ab

Step 5:)

Pop until (
Stack becomes empty
Postfix: ab+

Step 6: ^

Push ^
Stack: ^
Postfix: ab+

Step 7: (

Stack: ^(
Postfix: ab+

Step 8: (

Stack: ^((
Postfix: ab+

Step 9: c

Postfix: ab+c
Stack: ^((

Step 10: *

Stack: ^((*
Postfix: ab+c

Step 11: d

Postfix: ab+cd
Stack: ^((*

Step 12:)

Pop until (
Postfix: ab+cd*
Stack: ^(

Step 13: /

Stack: ^(/
Postfix: ab+cd*

Step 14: (

Stack: ^(/(
Postfix: ab+cd*

Step 15: e

Postfix: ab+cd*e
Stack: ^(/(

Step 16: -

Stack: ^(/(-
Postfix: ab+cd*e

Step 17: f

Postfix: ab+cd*ef
Stack: ^(/(-

Step 18:)

Pop until (
Postfix: ab+cd*ef-
Stack: ^(/

Step 19:)

Pop until (
Stack becomes: ^
Postfix: ab+cd*ef-/

Final Step: End of expression

Pop remaining operators
Stack: ^
Postfix: **ab+cd*ef-/^**

OR

(c) Write an algorithm to convert infix expression into postfix expression. (7 mark)

Introduction

An infix expression is written with operators placed **between operands** (for example: A + B).
However, postfix expressions place operators **after operands** (for example: A B +).
Postfix form is easier for computer evaluation as it **eliminates parentheses** and follows a strict order of execution.

To convert an infix expression into postfix, we use a **stack** to temporarily store operators and parentheses while scanning the expression from left to right. The final postfix expression contains all operands in their original order and operators in the order of evaluation.

Algorithm for Infix to Postfix Conversion (Using Stack)

Step 1: Start

Begin scanning the infix expression from left to right.

Step 2: If the scanned symbol is an operand

- Add it **directly** to the postfix expression.

Step 3: If the scanned symbol is a left parenthesis '('

- Push it onto the stack.

Step 4: If the scanned symbol is a right parenthesis ')'

- Pop operators from the stack and add them to postfix **until a left parenthesis '(' is encountered**.
- Discard both parentheses.

Step 5: If the scanned symbol is an operator (like +, −, , /, ^)

- While (stack is not empty **AND**
  top of stack has an operator with **greater or equal precedence**
  AND top operator is **not '('**):
  - Pop the operator from the stack and append to postfix.
- Push the current operator on the stack.

Step 6: After the full expression is scanned

- Pop all remaining operators from the stack
  and add them to the postfix expression.

Step 7: End

Final Notes (for exam writing)

- Precedence order used:
  ^ > * , / > + , −
- Associativity:
  - **Right associative:** ^
  - **Left associative:** +, −, *, /
- Parentheses must be handled carefully while popping.

Q.3 (a) Illustrate how stack is used in the recursion.(3 mark)

Introduction

Recursion is a programming technique where a function **calls itself** to solve a smaller part of a larger problem. Every time a recursive call is made, the computer must remember the **current state** of the function—its variables, return address, and intermediate data. To store this information, the system uses a special memory structure called the **stack**.

The stack plays a central role in recursion because it follows **LIFO (Last In, First Out)**, which perfectly matches how recursive calls return in reverse order.

How Stack Works in Recursion

1. **Each recursive function call creates an Activation Record (Stack Frame)**
   - The stack frame stores local variables, parameters, and the return address.
   - This frame is pushed onto the stack whenever the function calls itself.
2. **Stack grows deeper with each recursive call**

- As long as recursion continues, more and more stack frames get pushed.
- This represents moving deeper into the problem.

3. **When the base condition is reached**
    - No further recursive call is made.
    - The function starts to return.
4. **Stack unwinds in reverse (LIFO)**
    - The topmost stack frame is popped first.
    - Each returning function retrieves the previous state and continues processing.
5. **Final result is obtained after the stack becomes empty**
    - After all frames are popped, the recursion completes.

Example Explanation (Conceptual)

Consider a simple recursive function **factorial(n)**:

- Calling fact(4) causes calls:
  fact(4) → fact(3) → fact(2) → fact(1)

Textual diagram:

- Stack top: fact(1)
- Next: fact(2)
- Next: fact(3)
- Bottom: fact(4)

Once fact(1) returns, its frame is popped, then fact(2), and so on, until the final answer is produced.

(b) Describe Threaded Binary Tree with example.(4 mark)

Introduction

In a normal binary tree, many pointers remain **NULL** because every node can have at most two children.
For a tree with *n* nodes, there are exactly **n+1 NULL pointers**.
These NULL pointers can be used to make tree traversal faster without using recursion or an external stack.
A **Threaded Binary Tree** solves this problem by replacing NULL pointers with **threads** that point to the next node in the traversal order.

Definition

A **Threaded Binary Tree** is a binary tree in which the NULL left or right pointers are replaced with **threads**.
A thread acts as a link to the **in-order predecessor or successor**, allowing fast traversal without stack or recursion.

Types of Threaded Binary Trees

1. Single Threaded Binary Tree

- Only one type of pointer (either left or right) is threaded.
- If the right child is NULL, the right pointer points to the **in-order successor**.

2. Double Threaded Binary Tree

- Both left and right NULL pointers are replaced with threads.
- Left thread → in-order predecessor
- Right thread → in-order successor
- This allows complete in-order traversal efficiently.

Advantages of Threaded Binary Trees

- Eliminates the need for recursion in traversal.
- Eliminates the need for an explicit stack.
- Faster in-order traversal.
- Makes tree more space-efficient by using NULL pointers.

Example Explanation (Textual Diagram)

Consider the simple binary tree:

```
  20
 /  \
10   30
```

Normal Tree:

- 10's left pointer = NULL
- 10's right pointer = NULL
- 30's left pointer = NULL
- 30's right pointer = NULL

Threaded Binary Tree (In-order threading):

In-order sequence: **10 → 20 → 30**

- 10’s right NULL pointer becomes a thread pointing to **20**
- 30’s left NULL pointer becomes a thread pointing to **20**

Now, the tree behaves like:

```
  20
 /  \
```

10 30

\ /

(20) (20)

Here, the arrows in brackets denote **threads**, not child links.

(c) Write a C program for the following operations on a circular queue. 1. Insert 2. Delete 3. Display (7 mark)

```
#include <stdio.h>
#define SIZE 5

int cq[SIZE];
int front = -1, rear = -1;

// Insert element into circular queue
void insert(int item) {
    if ((front == 0 && rear == SIZE - 1) || (rear + 1 == front)) {
        printf("Queue is FULL\n");
        return;
    }

    if (front == -1) {      // first element
        front = rear = 0;
    } else {
        rear = (rear + 1) % SIZE;
    }
```

```
    cq[rear] = item;
    printf("Inserted: %d\n", item);
}

// Delete element from circular queue
void delete() {
    if (front == -1) {
        printf("Queue is EMPTY\n");
        return;
    }

    int item = cq[front];
    printf("Deleted: %d\n", item);

    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % SIZE;
    }
}

// Display elements of circular queue
void display() {
    if (front == -1) {
```

```
        printf("Queue is EMPTY\n");
        return;
    }

    printf("Circular Queue: ");
    int i = front;
    while (1) {
        printf("%d ", cq[i]);
        if (i == rear) break;
        i = (i + 1) % SIZE;
    }
    printf("\n");
}

int main() {
    insert(10);
    insert(20);
    insert(30);
    insert(40);
    insert(50);   // Queue full after this

    display();

    delete();
```

```
    delete();

    display();

    insert(60);
    insert(70);   // wrap-around insert

    display();

    return 0;
}
```

⭐ Explanation (Short for Exam)

Insert Operation

Check if queue is full.

If empty → front = rear = 0.

Otherwise rear = (rear + 1) % SIZE.

Insert element.

Delete Operation

Check if queue is empty.

Remove element at front.

If last element → reset front = rear = -1.

Otherwise front = (front + 1) % SIZE.

Display Operation

Start from front and print elements using circular increment.

OR

Q.3 (a) Write a recursive solution for Tower of Hanoi problem. (3 mark)

Introduction

The **Tower of Hanoi** is a classic problem used to demonstrate the concept of **recursion**. It consists of **three rods**—Source (S), Auxiliary (A), and Destination (D)—and **n disks** of different sizes placed on the source rod. The objective is to move all disks from the source rod to the destination rod following specific rules.

Rules of the Problem

1. Only **one disk** can be moved at a time.
2. A larger disk **cannot be placed on a smaller disk**.
3. Only the **top disk** of any rod can be moved.

Recursive Solution (Algorithm / C-style Logic)

TOH(n, S, A, D)

```
{
    if (n == 1)
    {
        move disk from S to D;
        return;
    }

    TOH(n-1, S, D, A);
    move disk from S to D;
    TOH(n-1, A, S, D);
}
```

Explanation

- If there is only **one disk**, it is directly moved from source to destination.
- For **n disks**, first move **n−1 disks** from source to auxiliary rod.
- Then move the **largest disk** to destination.
- Finally, move the **n−1 disks** from auxiliary to destination using recursion.

(b) Explain the DFS traversal of the graph with an example.(4 mark)

Introduction

Depth First Search (DFS) is a fundamental **graph traversal algorithm** used to explore all the vertices of a graph. In DFS, the traversal starts from a selected vertex and explores **as far as possible**

along one branch before backtracking. It uses either **recursion** or an explicit **stack** to keep track of vertices.

DFS is widely used in applications such as **path finding, cycle detection, and connected components analysis**.

Definition of DFS

Depth First Search is a traversal technique in which a graph is explored by visiting a vertex, then recursively visiting one of its unvisited adjacent vertices until no further expansion is possible, after which the algorithm backtracks.

Working of DFS Traversal

The DFS algorithm works in the following manner:

- Start traversal from a chosen **source vertex**.
- Mark the current vertex as **visited**.
- Visit an **unvisited adjacent vertex** and repeat the process.
- If no unvisited adjacent vertex exists, **backtrack** to the previous vertex.
- Continue until **all vertices** are visited.

Example Explanation (Textual Diagram)

Consider the following graph:

```
  A
 / \
B   C
```

/ \ \

D E F

DFS Traversal Starting from A

1. Visit **A**
2. Go to **B**
3. Go to **D** (no further child, backtrack)
4. Visit **E** (backtrack)
5. Backtrack to **A**
6. Visit **C**
7. Visit **F**

DFS Order

A → B → D → E → C → F

Advantages of DFS

- Requires **less memory** compared to BFS.
- Useful in **cycle detection**.
- Helps in solving **maze and path problems**.
- Suitable for **topological sorting**.

Limitations of DFS

- Does not guarantee the **shortest path**.
- Can go deep into recursion and cause **stack overflow** for large graphs.

(c) Write an algorithm to sort existing singly linked list in acceding order according to the information field. (7 mark)

Introduction

A **singly linked list** is a linear data structure in which each node contains two parts:

1. **Information field (data)**
2. **Link field (pointer to next node)**

Sorting a singly linked list means arranging the nodes in **ascending order based on the information field**. Since linked lists do not allow random access like arrays, sorting is generally done by **traversing nodes and swapping data** or by **changing links**. A simple and commonly used method is **Bubble Sort**, which is easy to understand and suitable for exam explanation.

Approach Used

- Traverse the linked list multiple times.
- Compare the information fields of **adjacent nodes**.
- Swap the data if the nodes are in the wrong order.
- Repeat the process until the list is completely sorted.

This approach does not change the node links, only the **data values**, making it simpler to implement.

Algorithm: Sort Singly Linked List in Ascending Order

Step 1: Start

Step 2: Check if the list is empty or has only one node

- If **head = NULL** or **head → next = NULL**, then the list is already sorted.
- Exit.

Step 3: Initialize pointer variables

- Set **ptr1 = head**
- Declare **ptr2**
- Declare a temporary variable **temp** for swapping data.

Step 4: Outer Loop (Repeat passes)

- Repeat until no swapping is required or until ptr1 reaches the last node.

Step 5: Inner Loop (Compare adjacent nodes)

- Set **ptr2 = head**
- While **ptr2 → next ≠ NULL**:
    - If **ptr2 → info > ptr2 → next → info**, then:
        - Swap the information fields of the two nodes.
    - Move **ptr2** to the next node.

Step 6: Continue passes

- Repeat the traversal until the entire list is sorted in ascending order.

Step 7: Stop

Pseudo-Code Representation

Algorithm Sort_Linked_List(head)

1. If head = NULL OR head->next = NULL

```
    return

2. Repeat
    swapped = false
    ptr = head

3. While ptr->next != NULL
    If ptr->info > ptr->next->info
        Swap(ptr->info, ptr->next->info)
        swapped = true
    ptr = ptr->next

4. Until swapped = false

5. End
```

Example Explanation

Consider a singly linked list:

Before Sorting:
30 → 10 → 20 → 40

Pass 1:

- Compare 30 and 10 → swap
- Compare 30 and 20 → swap
- Compare 30 and 40 → no swap

List becomes: **10 → 20 → 30 → 40**

Pass 2:

- No swaps occur → list is sorted.

After Sorting:
10 → 20 → 30 → 40

Advantages

- Simple and easy to implement.
- Does not require extra memory.
- Suitable for small-sized linked lists.

Disadvantages

- Time complexity is **O(n²)**.
- Not efficient for very large linked lists.

Q.4 (a) Define the following terms: 1. Field 2. Record 3. File (3 mark)

In data processing and data organization, information is stored in a **hierarchical manner** starting from the smallest unit to the largest. The terms **Field**, **Record**, and **File** are fundamental concepts used to organize data efficiently.

---

1. Field

- A **Field** is the **smallest unit of data** that represents a single piece of information.
- It stores **one specific attribute** of an entity.
- Examples include **Roll Number**, **Name**, **Age**, or **Salary**.

- Fields contain **atomic values**, meaning they cannot be further divided.
- Fields are the basic building blocks used to form records.

---

2. Record

- A **Record** is a **collection of related fields** that together describe a complete entity.
- It represents **one complete set of information** about an object or person.
- Example: A student record may include fields like Roll No, Name, Age, and Marks.
- Records are stored sequentially within a file.
- Each record corresponds to **one row** in a table.

---

3. File

- A **File** is a **collection of related records** stored permanently on secondary storage devices such as hard disks.
- All records in a file usually have the **same structure** and fields.
- Files are used for **data storage, retrieval, and processing**.
- Example: A student file contains records of all students in a class.

(b) Sort the following data using merge sort. 50, 20, 70, 05, 30, 80, 55, 25(4 mark)

Introduction

Merge Sort is a **divide and conquer** sorting algorithm. It works by dividing the given list into smaller sublists until each sublist contains only one element, and then **merging** these sublists in a sorted manner.

Merge sort is known for its **stable sorting** behavior and guaranteed time complexity of **O(n log n)**.

In this question, we will sort the given data step by step using the merge sort technique.

Given Data

Initial list:
50, 20, 70, 05, 30, 80, 55, 25

Total number of elements = **8**

Step 1: Divide the List (Divide Phase)

First Division

[50, 20, 70, 05] [30, 80, 55, 25]

Second Division

[50, 20] [70, 05] [30, 80] [55, 25]

Third Division

Each sublist is divided until single elements remain:

[50] [20] [70] [05] [30] [80] [55] [25]

At this stage, each element is considered **sorted**.

Step 2: Merge the Sublists (Merge Phase)

Merge Step 1

Merge adjacent single elements in sorted order:

[50] + [20] → [20, 50]

[70] + [05] → [05, 70]

[30] + [80] → [30, 80]

[55] + [25] → [25, 55]

Merge Step 2

Merge the resulting sorted sublists:

[20, 50] + [05, 70] → [05, 20, 50, 70]

[30, 80] + [25, 55] → [25, 30, 55, 80]

Final Merge Step

Merge the two large sorted lists:

[05, 20, 50, 70] + [25, 30, 55, 80]

Comparing elements step by step:

- Compare 05 and 25 → 05
- Compare 20 and 25 → 20
- Compare 50 and 25 → 25
- Compare 50 and 30 → 30
- Compare 50 and 55 → 50
- Compare 70 and 55 → 55
- Compare 70 and 80 → 70
- Remaining element → 80

Final Sorted List

05, 20, 25, 30, 50, 55, 70, 80

Key Features of Merge Sort

- Uses **divide and conquer strategy**
- Always divides array into two halves
- Stable sorting algorithm
- Requires additional memory for merging

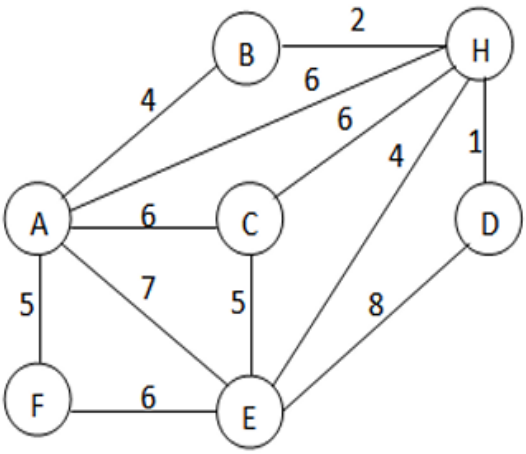
Advantages

- Guaranteed time complexity **O(n log n)**
- Suitable for large datasets
- Stable sort (maintains relative order of equal elements)

Disadvantages

- Requires extra memory
- Not an in-place sorting algorithm
- Slightly slower for small lists

(c) Find the minimum spanning tree of the following graph using Kruskal's algorithm.(7 mark)



Introduction

A **Minimum Spanning Tree (MST)** of a connected, weighted, undirected graph is a subgraph that:

- Includes **all the vertices**,
- Contains **no cycles**, and
- Has the **minimum possible total edge weight**.

Kruskal's Algorithm is a greedy algorithm used to find the MST. It works by selecting edges in **increasing order of weight**, ensuring that no cycle is formed at any step.

Steps of Kruskal's Algorithm

1. List all edges of the graph with their weights.
2. Sort the edges in **ascending order of weight**.
3. Select the edge with the **smallest weight**.
4. Add the edge to MST **if it does not form a cycle**.
5. Repeat steps 3 and 4 until **(n − 1) edges** are selected, where *n* is the number of vertices.

Step 1: List of Edges with Weights

Edge	Weight
H – D	1
B – H	2
A – B	4
H – E	4

Edge	Weight
A – F	5
C – E	5
A – H	6
A – C	6
C – H	6
F – E	6
A – E	7
D – E	8

Step 2: Sort Edges in Ascending Order

Edges are already arranged above in increasing order of weight.

Step 3: Select Edges (Avoiding Cycles)

Selected Edge	Reason
H – D (1)	Smallest weight
B – H (2)	No cycle
A – B (4)	No cycle
H – E (4)	No cycle
A – F (5)	No cycle
C – E (5)	No cycle

At this point:

- Number of vertices = **7**
- Required edges for MST = **6**
- All vertices are connected

So the MST is complete.

Minimum Spanning Tree Edges

- H – D (1)
- B – H (2)
- A – B (4)
- H – E (4)
- A – F (5)
- C – E (5)

Total Weight of MST

$$1+2+4+4+5+5=21$$

OR

Q.4 (a) What is hashing? Write the properties of a good hash function. (3 mark)

Introduction

Hashing is a technique used to **map data items (keys)** to fixed-size values called **hash values** using a special function known as a **hash function**. These hash values are used as **indexes in a hash table**, enabling fast data insertion, deletion, and searching. Hashing significantly reduces the time complexity of search operations.

Definition of Hashing

Hashing is the process of converting a given key into a **small integer value** using a hash function so that the data can be stored and retrieved quickly from a hash table.

Properties of a Good Hash Function

A good hash function should have the following properties:

1. **Uniform Distribution**
   - The hash function should distribute keys **evenly** across the hash table.
   - This minimizes collisions and improves performance.
2. **Deterministic**
   - For the same input key, the hash function must always produce the **same hash value**.
   - This ensures reliable searching and retrieval.
3. **Efficient Computation**
   - The hash function should be **simple and fast** to compute.
   - It should not increase the processing time unnecessarily.
4. **Minimize Collisions**
   - Different keys should ideally generate **different hash values**.
   - Fewer collisions lead to better performance.
5. **Uses All Parts of the Key**
   - The function should consider **all bits or characters** of the key.
   - This avoids clustering caused by similar keys.

(b) Sort the following data using quick sort. 50, 30,80,40, 35,70, 60, 20,75 (4 mark)

Introduction

Quick Sort is an efficient **divide and conquer** sorting algorithm. It works by selecting a **pivot element**, partitioning the list so that elements smaller than the pivot are placed on its left and larger elements on its right, and then recursively applying the same process to the sublists. Quick sort is widely used due to its **average time complexity of O(n log n)**.

Given Data

Initial list:
50, 30, 80, 40, 35, 70, 60, 20, 75

(Assume **first element as pivot** for explanation.)

Step 1: First Partition (Pivot = 50)

Elements smaller than 50 → left side
Elements greater than 50 → right side

30, 40, 35, 20 | 50 | 80, 70, 60, 75

Step 2: Sort Left Sublist (Pivot = 30)

Left sublist: **30, 40, 35, 20**

20 | 30 | 40, 35

Now sort remaining right part of this sublist:

Pivot = 40

35 | 40

Sorted left part becomes:

20, 30, 35, 40

Step 3: Sort Right Sublist (Pivot = 80)

Right sublist: **80, 70, 60, 75**

70, 60, 75 | 80

Now sort left part of this sublist:

Pivot = 70

60 | 70 | 75

Sorted right part becomes:

60, 70, 75, 80

Step 4: Combine All Sublists

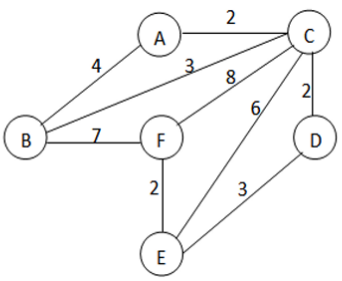
Final sorted order:

20, 30, 35, 40, 50, 60, 70, 75, 80

Final Sorted List

✅ **20, 30, 35, 40, 50, 60, 70, 75, 80**

(c) Find the shortest path from A to F using Dijkstra's Algorithm. (7 mark)



Introduction

Dijkstra's Algorithm is a greedy algorithm used to find the **shortest path from a single source vertex to all other vertices** in a weighted graph where all edge weights are non-negative. It works by repeatedly selecting the vertex with the minimum tentative distance and updating the distances of its neighboring vertices.

In this problem, Dijkstra's algorithm is applied to find the **shortest path from vertex A to vertex F**.

Graph Description (in words)

The given graph consists of six vertices: **A, B, C, D, E, and F**. The weighted edges are:

- A → C = 2
- A → B = 4
- B → C = 3
- B → F = 7
- C → F = 8
- C → E = 6
- C → D = 2
- D → E = 3
- E → F = 2

(All edges are assumed to be undirected.)

Step-by-Step Application of Dijkstra's Algorithm

Step 1: Initialization

- Source vertex = **A**
- Distance of A = 0
- Distance of all other vertices = ∞ (infinity)
- Mark all vertices as unvisited

Vertex Distance Previous

A	0	–
B	∞	–
C	∞	–
D	∞	–
E	∞	–
F	∞	–

Step 2: Visit Vertex A

- From A:
  - Distance to C = 0 + 2 = **2**
  - Distance to B = 0 + 4 = **4**

Vertex Distance Previous

A	0	–
C	2	A
B	4	A
D	∞	–
E	∞	–

Vertex	Distance	Previous
F	∞	–

Mark **A** as visited.

Step 3: Visit Vertex C (minimum distance = 2)

- From C:
  - To D = 2 + 2 = **4**
  - To E = 2 + 6 = **8**
  - To F = 2 + 8 = **10**

Vertex	Distance	Previous
A	0	–
C	2	A
B	4	A
D	4	C
E	8	C
F	10	C

Mark **C** as visited.

Step 4: Visit Vertex B (distance = 4)

- From B:
  - To F = 4 + 7 = 11 (not better than 10, so ignored)

No change in distances.

Mark **B** as visited.

Step 5: Visit Vertex D (distance = 4)

- From D:
  - To E = 4 + 3 = **7** (better than 8, update)

Vertex	Distance	Previous
E	7	D
F	10	C

Mark **D** as visited.

Step 6: Visit Vertex E (distance = 7)

- From E:
  - To F = 7 + 2 = **9** (better than 10, update)

Vertex	Distance	Previous
F	9	E

Mark **E** as visited.

Step 7: Visit Vertex F

- Destination reached with **minimum distance = 9**

Shortest Path Reconstruction

Tracing back from F using the “Previous” column:

F ← E ← D ← C ← A

Final Answer

✅ **Shortest Path from A to F:**

A → C → D → E → F

✅ **Total Shortest Distance:**

2 + 2 + 3 + 2 = 9

Q.5 (a) Compare linear search and binary search in terms of their time complexity. (3 mark)

Introduction

Searching is a fundamental operation in data structures used to find the position of a given element in a list. **Linear search** and **binary search** are two commonly used searching techniques. The efficiency of these searching methods is evaluated using **time complexity**, which represents the amount of time an algorithm takes relative to the size of the input.

Time Complexity of Linear Search

Linear search checks each element of the list **one by one** until the required element is found or the list ends.

- **Best Case:** O(1) – when the element is found at the first position
- **Average Case:** O(n) – element may be present anywhere in the list
- **Worst Case:** O(n) – element is at the last position or not present

Here, *n* represents the number of elements in the list.

Time Complexity of Binary Search

Binary search works by repeatedly dividing the sorted list into two halves and comparing the middle element with the target value.

- **Best Case:** O(1) – when the middle element is the key
- **Average Case:** O(log n)
- **Worst Case:** O(log n) – repeated halving until the element is found or search space becomes empty

Binary search is much faster but requires the data to be **sorted**.

Comparison Summary

- Linear search has **linear time complexity**, making it slower for large datasets.
- Binary search has **logarithmic time complexity**, making it more efficient.
- Binary search performs better than linear search as the input size increases.

(b) Write a C program for a bubble sort. (4 mark)

Introduction

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process is continued until the list is completely sorted. The algorithm gets its name because smaller elements gradually "bubble" to the top of the list.

Algorithm Idea (in words)

- Compare two adjacent elements.

- Swap them if the first element is greater than the second.
- Repeat the process for all elements.
- After each pass, the largest element moves to its correct position.
- Continue until no more swaps are required.

C Program for Bubble Sort

```
#include <stdio.h>

int main()
{
    int a[50], n, i, j, temp;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter array elements:\n");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }

    /* Bubble Sort Logic */
    for(i = 0; i < n - 1; i++)
    {
```

```
        for(j = 0; j < n - i - 1; j++)
        {
            if(a[j] > a[j + 1])
            {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }

    printf("Sorted array:\n");
    for(i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }

    return 0;
}
```

Explanation of Program

- The program accepts n elements from the user.
- Two nested loops are used:
    - Outer loop controls the number of passes.

    - Inner loop compares adjacent elements.
- Swapping is done using a temporary variable.
- After completion, the array is displayed in **ascending order**.

(c) What is hash collision? Explain collision resolution techniques. (7 mark)

Introduction

Hashing is a technique used to store and retrieve data efficiently using a **hash function**. A hash function maps a key value to an index (address) in a hash table. However, when two or more different keys produce the **same hash value**, a problem known as **hash collision** occurs. Since a hash table location can normally store only one element, special techniques are required to handle collisions.

Definition of Hash Collision

A **hash collision** occurs when **two or more distinct keys are mapped to the same index** in a hash table by a hash function.

In simple words, if:

$h(key1) = h(key2)$, where $key1 \neq key2$

then a collision has occurred.

Need for Collision Resolution

- Hash functions are not perfect.
- The number of possible keys is usually larger than the hash table size.
- Without collision resolution, data loss may occur.

- Efficient collision handling ensures fast searching, insertion, and deletion.

Collision Resolution Techniques

1. Separate Chaining

In this method, each index of the hash table stores a **linked list** of elements.

Explanation:

- All elements that hash to the same index are stored in a linked list.
- The hash table contains pointers to the head of the lists.
- New elements are added at the beginning or end of the list.

Features:

- Easy to implement.
- Table never gets full.
- Performance depends on the length of the chain.

Advantages:

- Simple and flexible.
- Handles large number of collisions.

Disadvantages:

- Requires extra memory for pointers.
- Searching becomes slower if chains become long.

---

2. Open Addressing

In this method, all elements are stored **within the hash table itself**. When a collision occurs, the algorithm searches for another empty location.

Open addressing has three common techniques:

(a) Linear Probing

- If a collision occurs at index i, the next index (i+1) is checked.
- This continues until an empty slot is found.

Example (in words):
If index 5 is occupied, check 6, then 7, and so on.

Advantages:

- Easy to implement.
- No extra memory required.

Disadvantages:

- Causes **primary clustering**.
- Performance degrades as table fills up.

(b) Quadratic Probing

- Uses quadratic intervals to find the next empty slot.
- Formula used:

h(key) + 1^2, 2^2, 3^2, ...

Advantages:

- Reduces primary clustering.
- Better than linear probing.

Disadvantages:

- Secondary clustering may still occur.
- Table may not be fully utilized.

(c) Double Hashing

- Uses a second hash function to calculate the new index.
- New index is calculated using:

(h1(key) + i * h2(key)) % table_size

Advantages:

- Minimizes clustering.
- Better distribution of keys.

Disadvantages:

- More complex to implement.
- Requires two good hash functions.

Comparison of Techniques (Brief)

- Separate chaining uses linked lists.
- Open addressing stores data directly in the table.
- Linear probing is simplest but least efficient.
- Double hashing provides best performance among open addressing methods.

OR

Q.5 (a) Does a pivot selection method affect the time complexity of quick sort? Justify your answer. (3 mark)

Introduction

Quick sort is a widely used **divide and conquer** sorting algorithm in which a **pivot element** is selected to partition the array into two sub-arrays. Elements smaller than the pivot are placed on the left side, and elements greater than the pivot are placed on the right side. The efficiency of quick sort greatly depends on how the pivot element is chosen.

Effect of Pivot Selection on Time Complexity

Yes, the **pivot selection method directly affects the time complexity of quick sort**. Different pivot choices result in different sizes of partitions, which influence the number of recursive calls.

- **Best Case:**
  When the pivot divides the array into two nearly equal halves, the time complexity is **O(n log n)**. This happens when the pivot is close to the median.
- **Average Case:**
  With a reasonably good pivot selection, quick sort generally performs in **O(n log n)** time.
- **Worst Case:**
  If the pivot is always the smallest or largest element, the partitions become highly unbalanced. In this case, the time complexity degrades to **$O(n^2)$**.

Justification (Example in Words)

If the first element is always chosen as a pivot and the array is already sorted, quick sort will repeatedly create one empty sub-array and one sub-array of size $(n-1)$. This results in maximum comparisons and hence worst-case performance.

(b) Write a C program for a selection sort. (4 mark)

Introduction

Selection sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the **smallest element** from the unsorted part of the list and placing it at the beginning. The algorithm divides the array into two parts: a sorted part and an unsorted part.

Working Principle (in words)

- Assume the first element is the minimum.
- Compare it with the remaining elements.
- Find the smallest element in the unsorted list.
- Swap it with the first unsorted position.
- Repeat the process until the list is fully sorted.

C Program for Selection Sort

```
#include <stdio.h>


int main()
{
   int a[50], n, i, j, min, temp;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter array elements:\n");
```

```
for(i = 0; i < n; i++)
{
   scanf("%d", &a[i]);
}

/* Selection Sort Logic */
for(i = 0; i < n - 1; i++)
{
   min = i;
   for(j = i + 1; j < n; j++)
   {
      if(a[j] < a[min])
      {
         min = j;
      }
   }

   /* Swap minimum element with first unsorted element */
   if(min != i)
   {
      temp = a[i];
      a[i] = a[min];
      a[min] = temp;
   }
```

```
    }

    printf("Sorted array:\n");
    for(i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }

    return 0;
}
```

Explanation of Program

- The program takes n elements as input.
- The outer loop selects the position where the minimum element should be placed.
- The inner loop finds the smallest element from the unsorted part.
- Swapping is done only once per pass.
- Finally, the sorted array is displayed in **ascending order**.

(c) List various file organizations and explain one in detail. (7 mark)

Introduction

A **file organization** refers to the method of arranging records in a file on secondary storage so that they can be accessed, retrieved, and updated efficiently. The choice of file organization affects the **performance**, **storage utilization**, and **processing speed** of file

operations such as searching, insertion, deletion, and updating of records.

List of Various File Organizations

The commonly used file organizations are:

1. **Sequential File Organization**
2. **Direct (Random) File Organization**
3. **Indexed File Organization**
4. **Indexed Sequential File Organization**
5. **Heap File Organization**

Sequential File Organization (Explained in Detail)

Definition

In **sequential file organization**, records are stored **one after another in a specific order**, usually based on a **key field** such as roll number or employee ID.

Working (Explanation in Words)

- Records are arranged in ascending or descending order of a key.
- To access a particular record, the file is read sequentially from the beginning until the desired record is found.
- New records are usually added at the end of the file or in a temporary overflow area.
- Updating or deleting records may require rewriting the entire file.

(If a diagram were drawn, it would show records stored linearly in order of key values.)

Features

- Simple structure and easy to implement.
- Records are stored in sorted order.
- Best suited for batch processing.

Advantages

- Easy to design and maintain.
- Efficient for sequential access.
- Requires minimal additional storage.

Disadvantages

- Slow for random access.
- Insertion and deletion operations are time-consuming.
- Not suitable for applications requiring frequent updates.

Applications

- Payroll processing
- Student result processing
- Bank statement generation

Brief Explanation of Other File Organizations

- **Direct (Random) File Organization:**
  Records are accessed directly using a hash function without searching sequentially.
- **Indexed File Organization:**
  Uses an index file that stores key values and pointers to actual records.
- **Indexed Sequential File Organization:**
  Combines features of both sequential and indexed file organizations.
- **Heap File Organization:**
  Records are stored in no particular order and are placed wherever space is available.
